I.J. Information Technology and Computer Science, **2020, 5, 14-37**
Published Online October 2020 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijitcs.2020.05.02

God Class Refactoring Recommendation and Extraction Using Context based Grouping

Tahmim Jeba
Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh
E-mail: bsse0533@iit.du.ac.bd

Tarek Mahmud[a], Pritom S. Akash[b] and Nadia Nahar[b]
[a] Department of Computer Science, Texas State University, San Marcos, Texas, USA
[b] Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh
E-mail: {bsse0508, bsse0604, nadia}@iit.du.ac.bd

Received: 18 February 2020; Accepted: 16 March 2020; Published: 08 October 2020

Abstract: Code smells are the indicators of the flaws in the design and development phases that decrease the maintainability and reusability of a system. A system with uneven distribution of responsibilities among the classes is generated by one of the most hazardous code smells called God Class. To address this threatening issue, an extract class refactoring technique is proposed that incorporates both cohesion and contextual aspects of a class. In this work, greater emphasis was provided on the code documentation to extract classes with higher contextual similarity. Firstly, the source code is analyzed to generate a set of cluster of extracted methods. Secondly, another set of clusters is generated by analyzing code documentation. Then, merging these two, a final cluster set is formed to extract the God Class. Finally, an automatic refactoring approach is also followed to build newly identified classes. Using two different metrics, a comparative result analysis is provided where it is shown that the cohesion among the classes is increased if the context is added in the refactoring process. Moreover, a manual inspection is conducted to ensure that the methods of the refactored classes are contextually organized. This recommendation of God Class extraction can significantly help the developers in minimizing the burden of refactoring on own their own and maintaining the software systems.

Index Terms*:* Code Smell, God Class, Extract Class Refactoring, Hierarchical Clustering, Cluster Composition, Automatic Refactoring.

1. Introduction

The term code smells in computer programming refer to the coding practices which reflect flaws in design and implementation phases that hinder code maintainability and understandability [1]. Although code smells do not cause technical inaccuracy, these indicate the flaws in the design and development practice. In Object Oriented design, it is highly expected that the job of the entities (i.e., classes, methods) would be defined and unifunctional. This approach is violated by one of the code smells named God Class. God Class or God Object refers to the scenario where a class does or knows more than it should according to the proper design concept. This uneven distribution of responsibilities among the classes makes the maintenance of whole system more complicated and difficult.

One of the most hazardous issues of God Class smell is increasing coupling and decreasing cohesion [1]. Poor design quality in terms of coupling and cohesion increases development and maintenance time, cost and human efforts, and makes the application complicated. It is also difficult to meet the change requirements in the highly coupled and loosely cohesive software applications [1]. As a result, it becomes very difficult for the developers to maintain the application in the long run. To enhance modularization of a system, extracting a God Class into smaller classes according to their specific functionalities has a significant effect. God Class identification and choosing appropriate refactoring is not easy as it requires significant experience and eort by the developers. Recommendation of extract class has two challenges. One is defining which classes are to be considered as God Class. And the second one is how to find the similarities among the methods. Here, context plays an important role in Object Oriented Design (OOD) which adds a dimension to find similarities in the refactoring research field. Context can be defined as the circumstantial explanation that helps to understand an event, statement or idea completely. In this research, the documented descriptions through the comment sections of the methods are considered to understand the context of a class. Contextual similarity refers that a method and a class are similar based on the context or the responsibility of the class.

Affluent number of researches have been conducted to refactor God Classes. Some of the existing researches split

classes based on coupling and cohesion, and others calculate the distance between the entities, and based on that, classes were extracted. In 2011 Bavota et al. proposed a two-step technique which supported extract class refactoring to define new classes with higher cohesion than the original class [2]. The cohesion between the methods was calculated using structural and semantic measures. Then, using the methods and their cohesion, a weighted graph was built, and the edges with lower cohesion than a predefined threshold were then cut. In 2013 as the extension of their previous work, they empirically evaluated the effectiveness of their tool on real Blobs from existing open source systems [3]. In another research, Gethers et al. proposed a cohesion calculation technique by combining Lack of Cohesion in Methods (LCOM) metric and conceptual similarity between methods [4]. They used the MaxFlow-MinCut Algorithm to split the weighted graph into two sub-graphs to build two new more cohesive classes. Oliveto et al. proposed an approach to support extract class refactoring systems that exploited the game theory technique in order to split a class into several classes with different responsibilities [5]. They cut the graph based on the minimum threshold of cohesion and MaxFlow-MinCut algorithm. However, since they did not consider the contextual similarities, the cut may contradict with the respect of context of the perspective classes. This issue leads to the objective of the proposed approach where one of the cohesion based solution [3] is integrated with contextual analysis.

Fokaefs et al. presented an extension for the JDedorant Eclipse plugin that employed a clustering technique to identify the extract class refactoring opportunities [6]. In this approach, the distance between the attributes and methods of each class was calculated using the Jaccard distance as the distance metric in the clustering procedure. Bavota et al. proposed another novel approach based on the Relational Topic Models (RTM) to recommend moving a class to a more suitable package to improve software modularization [7]. RTM was calculated using semantic and structural information. Using this analysis, the approach identified the possible move class refactoring opportunities to more suitable packages for relocating a class under analysis. However, these techniques calculated the distance between the class entities to extract classes or move the classes to another suitable package. If the contextual similarity can be accounted as one of the measuring dimensions, more accuracy can be achieved.

To explore this dimension, this paper proposes a refactoring technique that considers the contextual aspect of the class. For this refactoring approach, firstly an existing technique is implemented which is based on the cohesion calculation among the methods of the class [3]. Secondly, an approach is proposed which generates the clusters of the methods based on their documentation. To merge these two sets of clusters from two techniques, a cluster compositional algorithm is used which generates the final set of method clusters [8]. Each cluster of this set is supposed to be a new class. Therefore, the newly identified classes are extracted automatically from the source class and constructed as individual classes.

A comparative result analysis is provided where it is shown that the result becomes better if the context is added in the refactoring process. The technique is run on two open source software and the result is compared with the output of an existing system [3] which is considered as the base of this research. In terms of Conceptual Cohesion of Classes (C3) [9], the cohesion is 0.72 which is more than five times higher than pre-refactoring (0.13) and more than two times higher than the existing system (0.27). In terms of Lack of Cohesion of Methods (LCOM), the average LCOM in the refactored classes using proposed approach is 23 which is lower than both the pre-refactored classes (1,310) and the refactored classes using existing system (257). Moreover, contextual similarity among the methods of the refactored is manually inspected. The percentage of method misplace is also very low (on average 2.78%). Thus, along with being highly cohesive, the classes are assured of being contextually organized.

2. Literature Review

There exist several approaches which proposed different mechanisms and models to detect code smells [10,11,12,13]. Numerous studies introduced different approaches regarding God Class refactoring as well [2,3,6,7]. Since the purpose of this research is to refactor a God Class through clustering based on cohesion and context, this literature review section is presented with the existing studies related to refactoring techniques and clustering tools. The section is divided into the following major sections.

- **God Class Refactoring:** The researches regarding God Class Refactoring is studied thoroughly and described in this section.
- **Cluster Analysis:** If multiple refactoring solutions are needed to be incorporated together, a cluster merging algorithm is required to integrate multiple set of clusters of methods. Thus, the researches regarding cluster generation and applications are summarized in this section.

2.1. God Class Refatoring

Affluent number of researches have been conducted to refactor God Classes. The existing researches of God Class refactoring reflected some patterns. Most of the existing studies addressed the God Class issue in terms of cohesion and coupling. Thus, these studies intended to split classes based on cohesion calculation. These studies mainly focused on refactoring a God Class into smaller classes to make a system loosely coupled and highly cohesive. Others calculated distance between the entities, and based on that, classes were extracted. There are some more existing researches which

solved the problem by using Relational Topic Model (RTM), game theory, etc.

A. By Identifying Method Chains

Bavota et al. proposed several approaches to support extract class refactoring [2,3,14]. These approaches worked by identifying method chains to identify or recommend new classes with higher cohesion than the original class. The cohesion between two methods in a class was calculated using structural and semantic measures. A weighted graph was built considering the methods as nodes and their cohesions as edges. The methods of cutting this graph differed in different studies. In one of those researches, MaxFlow-MinCut algorithm was used to split the graph into two sub graphs to build two new classes [14]. However, this technique always allowed to extract a God Class into only two classes. In order to overcome that limitation they later proposed a two-step technique where the edges of the graph with lower cohesion than a predefined threshold were cut [2]. It was able to split the graph into sub-graphs by automatically identifying the appropriate number of classes that should be extracted from a God Class. In 2013, as the extension of their previous work, they empirically evaluated and assessed the effectiveness of their approach on real Blobs from existing open source systems [3]. Akash et al. also proposed an approach for extracting a God Class using both structural and semantic relationship between methods in that class [15]. In this approach, they introduced a new idea for representing methods in vector space using Latent Dirichlet Allocation (LDA) [16]. However, along with structural and semantic measures, there remains a scope of considering the contextual perspective of the class entities. Exploring this scope is considered as the objective of this proposed research.

B. Using Clustering Technique

In this section, different techniques are mentioned which used clustering algorithm to extract God Classes. Both of those used Hierarchical Agglomerative Clustering (HAC) [17, 18].

JDeodorant is a popular eclipse plugin that identifies five kinds of code smells, namely - Feature Envy, Duplicate Code, Type Checking, Long Method and God Class [6, 19, 20, 21]. The tool identifies extract class opportunities by applying a HAC on a God Class [6]. JDeodorant used the Jaccard distance [22] as the distance metric instead of Euclidean distance. In this iterative process, at each step, the algorithm merged the two closest clusters according to the single-linkage criterion. When all the clusters were more distant to each other than a predefined distance threshold, no more clusters could be merged and thus, the process stopped. The algorithm was run for different values of the threshold ranging from 0.1 to 0.9 with a 0.1 step. A single fixed threshold was not sufficient to obtain all the possible clusters that were produced by the hierarchical clustering algorithm. The effectiveness of the tool had been evaluated on the JHotDraw system. In 12 cases (75%), the evaluator confirmed that the classes suggested to be extracted indeed described a separate concept. In 9 cases (56%), the expert agreed that he would perform the refactoring according to the suggestions of the tool.

In another research, Fokaefs et al. used the same clustering algorithm to identify the opportunities of extracting God Classes [23]. They applied their methodology on two projects (eRisk and SelfPlanner) and asked the designers to give their feedback. For both projects, it was able to identify a relatively large number of new concepts (75.6% and 86% respectively) that can be potentially extracted in new classes.

Jiang et al. proposed a Large Class bad smell detection approach based on scale distribution [24]. In this research, a new model was developed which used class length distribution model and cohesion metrics to detect the smell. Moreover, the scheme of Extract Class is also proposed for refactoring the Large Class using Agglomerative Clustering Technique.

However, in these techniques, only structural information was taken into account to perform extract class refactoring. Finding the accurate distance threshold, is also a difficult problem.

C. Other Approaches

G. Bavota et al. proposed a novel approach based on Relational Topic Models (RTM) [25] to recommend Extract Class refactoring operations aiming at moving a class to a more suitable package to improve software modularization [7]. For this, the RTM was computed using two factors (structural and semantic information) extracted from the source code. Using the results of the analysis, the approach was able to identify the possible move class refactoring opportunities. In another paper, Bavote et al. also proposed an approach based on game theory to support extract class refactoring opportunities [5]. Given a class to be refactored, this approach modeled a non-cooperative game where two players contend for the methods of the original class to build two new classes with higher cohesion than the original class. However, the approach is a semi-automated system because it takes as input a class previously identified by the software engineer as a candidate for the refactoring.

All the refactoring techniques end up with producing a set of clusters of methods which are supposed to be in the newly identified classes. There are some scenarios where multiple solutions are needed to be incorporated together for a better solution. Therefore, there should be an approach which takes solutions of all the techniques and incorporates those to generate a cumulative solution which would be more significant according to the requirements. This issue leads to the further study regarding cluster analysis.

D. Using Clustering Technique

Numerous researches have been conducted regarding the cluster generation algorithm and their application. K-means clustering algorithm, a subset of unsupervised learning, can group a dataset into k number of categories [17]. Here, k indicates the number of clusters which are represented by their centroids. The principle is to minimize the sum of squared distances between data and the corresponding cluster centroids. Kanungo et al. presented a simple and efficient implementation of the K-means clustering algorithm called *filtering algorithm* [26]. In the basic k-means algorithm, at each stage, data points are assigned to its closest centroid and the centroid of each cluster is recomputed. Using kd-tree [27] data structure, the computation is reduced. Kd-tree is built once for the given data points. Since the data points are not changed in the process of computation, the tree is not needed to be recomputed at each iteration.

Unlike k-means clustering algorithm, K-Nearest Neighbor (KNN) is a classification algorithm which is a subset of supervised learning algorithm [28]. In this algorithm, a training dataset is given which are divided into some classes. Here, a number k is also given which represents the number of neighbors that are supposed to influence the classification. When a new test sample is given to classify, firstly distances with all the test data are calculated. The given data is allocated to the class of majority of its closest or nearest neighbours.

A Hierarchical Agglomerative Clustering (HAC) represents a group of closely related clustering technique [17]. All the points from the given dataset is initially considered as clusters and the distance between every two of those are calculated. In each iteration, the technique merges two closest clusters into one and the distance is updated. The hierarchy of clusters is demonstrated in a graphical representation called dendogram. An HAC is used in God Class refactoring [6, 29]. In these researches, the methods of a God Class are considered as the clusters, and Jaccard distance is used to calculate similarity between those. The technique repeatedly merges two clusters with higher similarity and the process stops when no other clusters remain left to be merged. Anquetil et al. presented a comparative study where different aspects of the clustering procedures were presented [30]. The parameters and their effects in clustering results, while doing software re-modularization with agglomerative hierarchical clustering algorithms, were analyzed in this research.

There are some other clustering algorithms, e.g., density based clustering (DBSCAN) [17], Mean-Shift Clustering [31], etc. All these techniques result in producing a set of clusters. However, if the results of any two techniques are needed to be incorporated together, there would be a problem of comparing these two sets to get a more appropriate set of clusters. In authors' previous work, a cluster compositional algorithm was proposed where multiple individual sets of clusters were assimilated into one [8]. This two phased compositional algorithm dealt with multiple sets of clusters generated from same dataset using different techniques to incorporate those into a more appropriate set of clusters. Firstly, the approach compared all the clusters from both input sets of clusters thoroughly to generate a cumulative set of clusters. In the second phase, the approach analyzed the newly identified sets to find out if there exists any single element clusters and finally merged those with other clusters from the set.

This cluster compositional algorithm [8] has the potential to play a vital role in God Class refactoring where the classes are considered as the clusters and the methods are considered as the element of those clusters. If a God Class is extracted into more than one set of smaller classes using different refactoring techniques, the cluster compositional algorithm can be applied to get an appropriate single set of refactored classes.

3. Proposed Methodology

The whole procedure of extracting a God Class is explained briefly in this section. As mentioned in the Section 2, the existing researches intend to solve the problem based on cohesion calculation, distance among the entities Relational Topic Model (RTM), and game theory etc. None of those consider integrating context with those solutions. However, considering the context of the class and the methods can add positive dimensions to the God Class extraction.

Proposed methodology has three integral parts (demonstrated in Fig.1.). First, the source code is analyzed to generate a cluster set of extracted methods [3]. In the next phase, another set of clusters is generated by analyzing code documentation. Then, merging these two sets, a final cluster set is formed to extract the God Class. Finally, an automatic refactoring approach is also followed to build newly identified classes.

In Fig.1., the procedure of refactoring starts with receiving a God Class as input. Then, the class is analyzed following two phases - *source code analysis* and *documentation analysis*. In the first phase, three measures: Structural Similarity between Methods (SSM), Call-based Interaction between Methods (CDM) and Conceptual Similarity between Methods (CSM) (discussed in the section 3.1) are incorporated and produced a set cluster, *Cluster A* [3]. Next, the code documentation of the same class is analyzed and using the HAC, another set of clusters, *Cluster B* is produced. Using cluster compositional algorithm [8], these two sets are merged and a final cluster is formed to extract the God Class. Finally, an automatic refactoring approach is also followed to build newly identified classes.

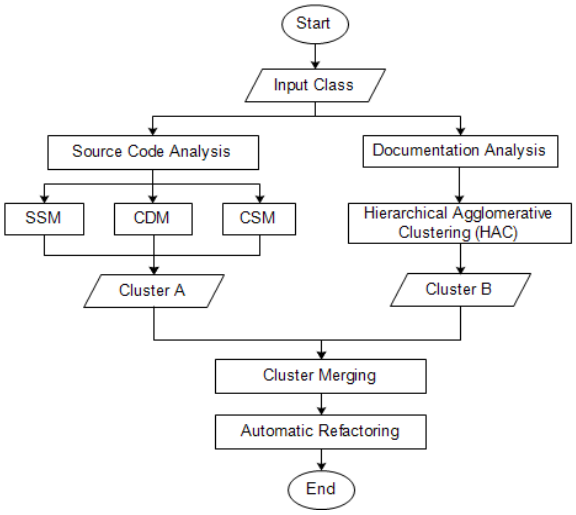


Fig.1. Overview of the Proposed Methodology

3.1. Source Code Analysis

In this process a class that is identified as a God Class is taken as input by the system [3]. The cohesion among the methods are calculated and stored in an $n \times n$ matrix. Here, n is the number of the methods of the class that has to be refactored. Three different measures are used to capture how the methods are relatable to each other. Using these measures firstly, the cohesion of each possible pair of methods is calculated. Next, a weighted graph is built where each node represents a method, and the weight of an edge that connects two nodes is given by the cohesion of the two methods. The higher the cohesion between two methods the higher the likelihood that the two methods should be in the same class. A cohesion threshold is applied to cut all the edges having cohesion lower than the threshold in order to find the method clusters for the new classes.

A. Cohesion Calculation

Three different measures [3], i.e., Structural Similarity between Methods (SSM), Call-based Interaction between Methods (CIM) and Conceptual Similarity between Methods (CSM) are used to calculate the cohesion of a class. These measures capture three distinct ways in which methods relate to one another, each reflecting a different type of relationship between methods.

The first measure is SSM which calculates cohesion by shared attribute reference of a method pair [3]. The calls performed by the methods of the class is considered in the second measure named CIM [3]. Finally, CSM of two methods is calculated as the cosine of the angle between their corresponding vectors [3]. In the existing technique, CSM calculation covers the source code and the comments contained in it [3]. Since among three, only one measure (CSM) takes comments under consideration, documentation does not get enough impact in the whole refactoring procedure. However, the proposed approach intends to capture the contextual aspect of a God Class from the documentation of the methods and give an equal emphasis. For this, source code and documentation analysis are divided into two different sections. CSM calculation only depends on source code while documentation analysis is separated from the CSM and considered as an individual branch of this proposed approach (Fig.1.). Thus, both the source code and documentation of a class are given equal emphasis in refactoring procedure. Lastly, the final cohesion among the methods are calculated by the weighted summation of these three measures (SSM, CIM, and CSM). To determine the weights Principal Component Analysis (PCA) of the values of these three measures computed on all the classes of the system is performed [3]. The value of the proportion of variance obtained for each measure is used as the weight for the corresponding measure [3].

B. Method Cluster Identification

In this phase, a weighted graph is built where each node represents a method, and the weight of each edge presents the cohesion of the two methods connected by the edge. The matrix is then filtered based on a minimum cohesion threshold. The edges of the graph with lower cohesion than the threshold were cut.

Thus, a number of method clusters are found that should be extracted from a God Class. Since classes with a very low number of methods do not have enough impact to exist, classes with lower number of methods than a predefined threshold are identified as trivial chains. The median of the non-zero values of the method-by-method matrix is used to define the threshold [3]. These identified trivial chains are then integrated with most cohesive non-trivial chains [3]. Thus, a set clusters is generated with the methods of the God Class based on source code analysis.

3.2. Documentation Analysis

While extracting a God Class, the proposed research intends to consider the contextual aspect of the class. Thus, the analysis of code documentation is also assembled along with the source code analysis. The purpose of the documentation analysis is to capture the context of the class and its method. The HAC algorithm is used to understand the contextual similarity between the methods and make clusters of those methods based on the similarity measure.

The process of documentation analysis is demonstrated in Fig.2. In this process, first the documented description of the methods is parsed and normalized. Secondly, a matrix containing a similarity measure between the method pairs of the class is generated. Finally, based on that similarity matrix, an iterative clustering process is followed. The iteration continues until no method is left isolated. The whole cluster process based on documentation is described in the following sections.

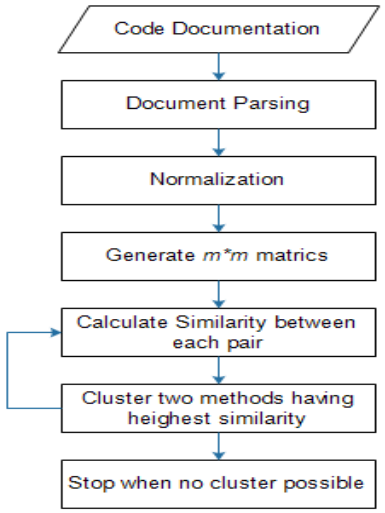


Fig.2. Overview of Code Documentation using Clustering

A. Data Pre-processing

Before starting the clustering procedure, data is needed to be pre-processed. This pre-processing includes identifying the description part for a particular method, parsing the words from the document and normalizing those words.

Document Parsing: The first step of this phase is to identify the part of the document which is considered as the description of a particular method. Both the comment section written exactly before a method and the comment lines in the method body are considered as the description of that method. For example, in Fig.3., the marked parts are the document section for a method, *comment* [32]. The text of the identified sections are parsed and words are stored with the corresponding method for the further procedure.

```
/**
 * A comment.
 *
 * @param text The text in the comment.
 * @param augs     Additional information that may include infoset augmentations
 *
 * @throws XNIException Thrown by application to signal an error.
 */
public void comment(XMLString text, Augmentations augs) throws XNIException {

    try {
        // SAX2 extension
        if (fLexicalHandler != null) {
            fLexicalHandler.comment(text.ch, 0, text.length);
        }
    }
    catch (SAXException e) {
        throw new XNIException(e);
    }

}
```

Fig.3. Documentation Section of a Method

Normalization: The parsed document of each method is normalized by using an Information Retrieval (IR) normalization process [33]. Thus, the terms extracted from the code documentation are normalized by applying the following steps [34]:

- Splitting the terms using the camel case splitting which splits words based on underscores, capital letters and numerical digits.
- Converting the letters of those words to lower case.
- Removing special characters, programming keywords and common English stop words.
- Stemming words to their original roots via Porter's stemmer [35]. Using this stemmer the suffix and prefix of all words are removed and the words are transferred into their root forms.

After following those steps, weights are added to the normalized words using the term frequency - inverse document frequency (tf-idf) [33]. These values help in decreasing the relevance of excessively generic words contained in most source segments.

B. Similarity Matrix Construction

To produce the clusters of methods using Hierarchical Agglomerative Clustering (HAC), a similarity measure is required. Here, cosine similarity is considered for that purpose. A m*m matrix is generated where the similarity measures among the method pairs is stored. Using the value of tf-idf, similarity between two methods are calculated as the cosine of the angle between their corresponding vectors.

$$CSM(m_a, m_b) = \frac{\overrightarrow{m_a} . \overrightarrow{m_b}}{\left\|\overrightarrow{m_a}\right\| . \left\|\overrightarrow{m_b}\right\|} \quad (1)$$

Where $\overrightarrow{m_a}$ and $\overrightarrow{m_b}$ are the vectors corresponding to the methods m_a and m_b.

	M1	M2	M3	M4	M5
M1	0				
M2	.23	0			
M3	.22	.35	0		
M4	.37	.20	.15	0	
M5	.36	.14	.28	.29	0

Fig.4. Example of a Similarity Matrix

In Fig.4., an example is shown where five methods are considered in a class - M1, M2, M3, M4 and M5. The values denote the cosine similarities among the method pairs. For example, cosine similarity between M1 and M2 is 0.23. If the identical elements are paired, the similarity is zero (e.g. cosine similarity between M1 and M1 is 0).

C. Adapted Hierarchical Agglomerative Clustering (HAC)

The purpose of this phase is to generate a set of clusters of the methods from the God Class based on their contextual similarities. To achieve this, the HAC algorithm is adapted in the proposed approach.

This algorithm starts by assigning each class member to a single cluster. In each iteration, it merges the two closest clusters and the similarity matrix is updated accordingly. Finally, the algorithm terminates when all the entities are contained in a single cluster, which forms the root of a hierarchy of clusters. The actual clusters can be determined at the merging points. The hierarchy of the clusters is usually graphically represented by a dendogram. The root is the final cluster (in this research, containing all the methods, it forms the input God Class again). The leaves of the tree represent the entities, and the intermediate nodes are the actual clusters. The height of the tree represents the different levels of the distance in which two clusters were merged. Some considerations for adopting the algorithm in the approach are listed below:

- In traditional Agglomerative Hierarchical Clustering algorithm, Eucledian distance is used as the distance measure among every two nodes. However, the methods are considered as the nodes in this research and cosine similarity is used in this research to figure out how the methods are related to each other.
- Since similarity measures among the method pairs are considered, at each iteration of the clustering, the two clusters having highest similarity (instead of lowest distance) are merged. After each iteration the similarity matrix is updated.
- To determine the actual set of clusters, a threshold value for the maximum similarity as a cut-off value, is needed to be chosen. In this methodology, three different ranges applied 25% (100/4), 33% (100/3) and 50% (100/2) as the cut-off value to find out the more appropriate one. From the experiment of this research, if the

values less than 25% are considered, very granular level clusters with lower number of methods are found. Again, values over than 50% make the clusters very large (sometimes it forms the original class). Therefore, the ranges over 50% and under 25% are avoided from the consideration.

D. An Illustrative Example

To better understand, let's explain this clustering with an example. Suppose, there are five methods in a God Class: M1, M2, M3, M4 and M5. The cosine similarity among the pairs of the methods are shown in the Fig.4. The iterations of the process is described here:

Iteration 01: In Fig.5(a)., 0.37 is the highest value in the matrix. Thus, the two methods M1 and M4 are merged into a cluster. In Fig.5(b)., to update the table, the columns of these two methods are merged and the value is changed accordingly. For example, when the new similarity value of (M1, M4) and M2 is updated, both similarity values (M1, M2) and (M4, M2) are compared (the values are 0.23 and 0.20) and the maximum value 0.23 is selected to update the cell. The other two updates (with M3 and M5) are shown in the Fig.5(b).

	M1	M2	M3	M4	M5
M1	0				
M2	.23	0			
M3	.22	.35	0		
M4	.37	.20	.15	0	
M5	.36	.14	.28	.29	0

(a) Merge most similar two clusters M1 and M4

Update similarity matrix

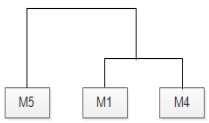
- MAX (sim (M1,M4), M2)
  = MAX (sim (M1,M2), (M4,M2))
  = MAX (.23,.20)
  = .23
- MAX (sim (M1,M4), M3)
  = MAX (sim (M1,M3), (M4,M3))
  = MAX (.22,.15)
  = .22
- MAX (sim (M1,M4), M5)
  = MAX (sim (M1,M5), (M4,M5))
  = MAX (.36, .29)
  = .36

	M1,M4	M2	M3	M5
M1,M4	0			
M2	.23	0		
M3	.22	.35	0	
M5	.36	.14	.28	0

(b) Update similarity matrix

Fig.5. First Iteration of Clustering process

Iteration 02: In this iteration, (M1, M4) and M5 are the closest clusters since these two have highest similarity value 0.36. Thus, these two nodes are merged (Fig.6(a).). The matrix is needed to be updated again. The update procedure is shown in Fig.6(b).



	M1,M4	M2	M3	M5
M1,M4	0			
M2	.23	0		
M3	.22	.35	0	
M5	.36	.14	.28	0

(a) Merge most similar two clusters (M1, M4) and M5

Update similarity matrix

- MAX (sim (M1,M4,M5), M2)
  = MAX (sim ((M1,M4), M2), (M5,M2))
  = MAX (.23,.14)
  = .23
- MAX (sim (M1,M4,M5), M3)
  = MAX (sim ((M1,M4), M3), (M5,M3))
  = MAX (.22,.28)
  = .28

	M1,M4,M5	M2	M3
M1,M4,M5	0		
M2	.23	0	
M3	.28	.35	0

(b) Update similarity matrix

Fig.6. Second Iteration of Clustering process

Iteration 03: In this iteration, the similarity between M2 and M3 is highest (0.35) and so, these two are merged (Fig.7(a).). The matrix is needed to be updated again. The update procedure is shown in Fig.7(b).

Iteration 04: Only two nodes are left to be merged. The similarity between M2 and M3 is highest (0.28) and so, these two are merged. Since no other nodes are left to be merged, the step of matrix update is not applicable here. The fourth and final iteration is shown in Fig.8.

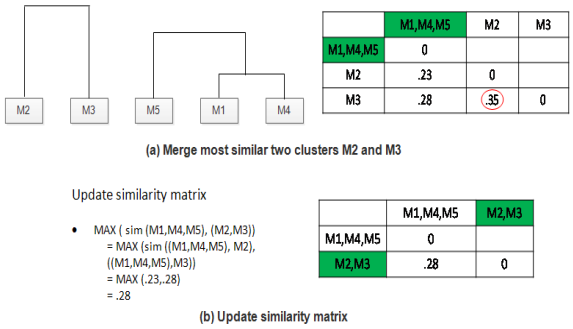


Fig.7. Third Iteration of Clustering process

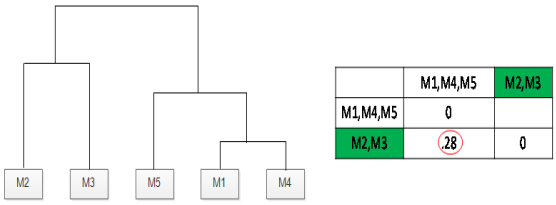


Fig.8. Final Iteration of Clustering process

The process of clustering is stopped here because all the nodes are now connected. If the final dendogram is observed, the number of clusters varies in different layers (shown in Fig.9.). For example, when 50% is considered as the cut-off value and the similarity values are filtered by this upper-limit, the set of clusters is: (M2, M3), (M5, M1, M4). For cut-off value 33%: (M2), (M3), (M5, M1, M4) and for cut-off value 25%: (M2), (M3), (M5), (M1, M4). In this research, three of these values (50%, 33% and 25%) are used while running the approach on the dataset. The outcome is better if 33% is used among three of those. Therefore, 33% is considered as the cut-off value in the proposed approach.

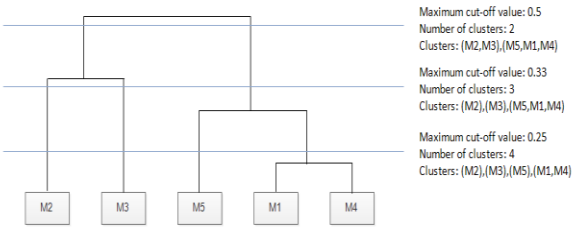


Fig.9. Final Dendogram Showing Set of Clusters for Different Threshold

3.3. Application of Cluster Compositional Algorithm

A God Class is extracted following both the source code analysis (Section 3.1) and the documentation analysis (Section 3.2) respectively. After completing those two stages, two sets of clusters of refactored classes are generated. Now, there is an issue of comparing and incorporating these two sets to generate a single set of clusters which is more convenient. Each cluster of these set represents a class that are supposed to be extracted from the original God Class. In this section in order to incorporate two sets of classes, first their methods are compared and then, cluster compositional algorithm [8] is used to merge.

To explain the scenario, let's consider an example of God Class with the methods: *m1*, *m2*, *m3*, *m4*, *m5*, *m6* and *m7*. First the class is refactored following the source code analysis based on cohesion calculation and the refactored classes are generated *(m1,m2), (m3,m4,m5), (m6,m7).* Secondly, the same God Class is refactored by code documentation analysis and the produced list of refactored classes is (m1,m2), (m3,m4,m6), (m5,m7). Now, these two sets of classes are required to be compared to produce a more relevant set of classes. To accomplish this, the two phased cluster compositional algorithm [8] is followed. First, all the classes from both sets are thoroughly compared and matched to generate a cumulative set of classes. In this new set, some classes might exist with only one method which drives the process in the next stage where the classes with single method are merged.

A. Comparing Two Groups of Classes

In the first stage of this compositional algorithm, the classes of two sets generated by two previous techniques (Section 3.1 and Section 3.2) are thoroughly compared and merged into a cumulative set. Let's consider *Group A* and *Group B* are the sets of classes and *Group F* is considered as the final set to store the merged set of classes. The classes of *Group A* is compared with the classes of *Group B*. At each iteration, a class from Group A is matched to the classes

of Group B one by one and the common methods are moved to the final set *Group F* as an individual class.

Let's explain this process with the help of previous example. Initially the recommended refactored classes are *Group A: (m1,m2), (m3,m4,m5), (m6,m7)*; *Group B: (m1,m2), (m3,m4,m6), (m5,m7)*; and *Group F: ø*. At first iteration, (m1,m2) from *Group A* is matched with (m1,m2), (m3,m4,m6), (m5,m7) from *Group B* one by one. (m1,m2) is identically matched with the first class of *Group B* and thus, pushed to *Group F* as a class (Fig.10(a).). No other method is left to match with the methods of the rest two classes (m3,m4,m6) and (m5,m7) (Fig.10(b), (c)). The first iteration ends here and *Group F*: (m1,m2) is recorded.

Table 1. The Name and LOC of the Considered God Classes of *Xerces*

Class Name	LOC
AbstractDOMParser	1775
AbstractSAXParser	1360
BaseMarkupSerializer	1275
CoreDocumentImpl	1497
DeferredDocumentImpl	1612
DOMNormalizer	1291
DOMParserImpl	820
DurationImpl	1998
NonValidatingConfiguration	783
XIncludeHandler	1331

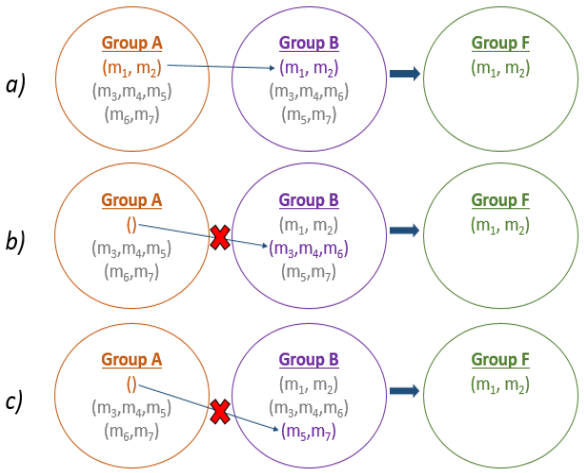


Fig.10. Comparing Two Groups of Classes: *Iteration 01*

At the next iteration, (m3,m4,m5) from *Group A* is again matched with (m1,m2), (m3,m4,m6), (m5,m7) from *Group B* one by one. Firstly, no matches found with the methods of the first class of *Group B* (Fig.11(a).). However, the (m3,m4,m5) from *Group A* is partially matched with the second class (m3,m4,m6) of *Group B*. The common methods (m3,m4) of the both classes are moved to *Group F* (Fig.11(b).) and (m5) is left to match with (m5,m7). (m5) is matched in both classes and moved to *Group F* (Fig.11(c).). Thus, after the second iteration, the final set of class becomes *Group F: (m1,m2), (m3,m4), (m5).*

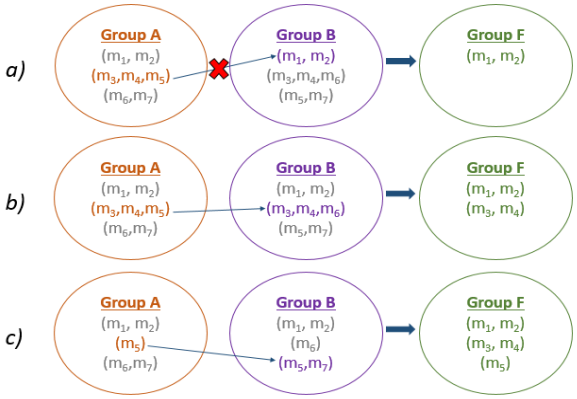


Fig.11. Comparing Two Groups of Classes: *Iteration 02*

Table 2. C3 Value Comparison Shown on God Classes of *Xerces*

Class	Pre-refactoring	Existing Approach		Proposed Approach	
		NOC	C3	NOC	C3
AbstractDOMParser	0.21	2	0.25, 0.23	2	0.64, 0.47
AbstractSAXParser	0.09	3	0.22, 0.29, 0.19	3	0.82, 0.70, 0.91
BaseMarkupSerializer	0.08	2	0.18, 0.15	4	0.73, 0.78, 0.54, 0.42
CoreDocumentImpl	0.05	3	0.19, 0.33, 0.12	8	0.78, 0.95, 0.97, 0.70, 0.67, 0.73, 0.34,0.79
DeferredDocumentImpl	0.14	2	0.18, 0.20	6	0.76, 0.94, 0.68, 0.82, 0.54, 0.44
DOMNormalizer	0.08	2	0.33, 0.17	4	0.57, 0.63, 0.58, 0.68
DOMParserImpl	0.24	2	0.38, 0.33	3	0.50, 0.41, 0.92
DurationImpl	0.11	2	0.22, 0.18	2	0.71, 0.97
NonValidatingConfiguration	0.04	2	0.31, 0.08	2	0.67, 0.78
XIncludeHandler	0.08	4	0.42, 0.14, 0.22, 0.27	7	0.74, 0.73, 0.52, 0.62 0.96, 0.90, 0.85

Table 3. The Name and LOC of the Chosen God Classes of *GanttProject*

Class Name	LOC
GanttOptions	513
GanttProject	2269
GanttGraphicArea	2160
GanttTree	1730
GanttTaskPropertiesBean	919
ResourceLoadGraphicArea	1060
TaskImpl	437

Moving to the third iteration, now (m6,m7) is compared with (m1,m2), (m3,m4,m6), (m5,m7) and does not match with the first class (m1,m2) (Fig.12 (a).). From the second classes of the both sets (f) is matched and pushed to *Group F* (Fig.12(b).). (m7) and (m5,m7) are matched and (m7) is moved to *Group F* (Fig.12(c).). Since no other class is left in *Group A*, the third iteration stops here and the final *Group F: (m1,m2), (m3,m4), (m5), (m6), (m7)* is found.

By using the first stage of the compositional algorithm [8] two sets of classes are merged into a single set of classes. The classes of this new set are supposed to be the extracted from the given God Class. However, three classes (m5), (m6), (m7) exist in this list with single methods which are required to be merged.

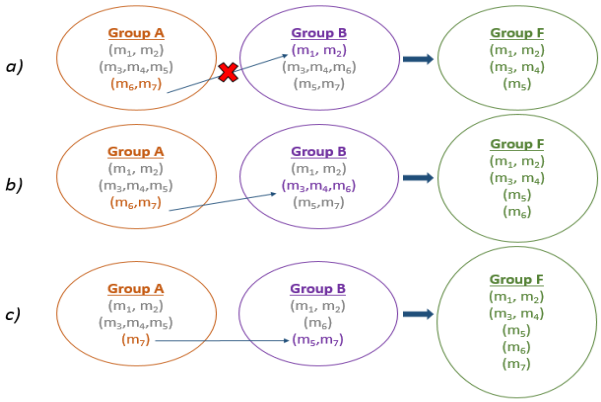


Fig.12. Comparing Two Groups of Classes: *Iteration 03*

B. Merging Classes with Single Method

A class with a single method does not have enough impact and it is highly expected to merge these classes with appropriate another ones. Therefore, the second phase of the cluster compositional algorithm [8] is to merge such classes with other classes of the set. For this merging procedure, the algorithm requires distance or similarity calculation between the class with single method and the rest of the classes [8]. In this research, to avoid the result being biased, a customized similarity matrix is constructed by the weighted summation of both the cohesion matrix from source code analysis (Section 3.1.1) and the cosine similarity matrix from code documentation analysis (Section 3.2.2). Therefore, the formation of the new matrix is:

$$\begin{aligned} &\text{New Similarity Matrix} \\ &= w_S.\text{Cohesion Matrix of Source Code} \\ &+ w_D.\text{Cosine Similarity Matrix of Documentation} \end{aligned} \tag{2}$$

Where $w_S + w_D = 1$ and their values express the weight in each measure and for this work, the both weights are considered equal. Based on this similarity measure a comparison is performed between the single left method with all other methods from the other classes. The average similarity is calculated between those to decide where the method will fit through merging.

Let's consider the example of the God Class again to demonstrate the merging process. From the previous step, the final set of classes was *Group F: (m1,m2), (m3,m4), (m5), (m6), (m7).* In this group of classes, three classes (m5), (m6), (m7) exist with single method and required to be merged with others one by one. At the first iteration, (m5) is considered to be merged and thus, the average similarity between this method and the methods from other four classes *(m1,m2), (m3,m4), (m6), (m7)* are needed to be calculated. The calculation is shown below:

$$\text{avgSim}_1 = \frac{\text{Sim}(m_1, m_5) + \text{Sim}(m_2, m_5)}{2} \tag{3}$$

$$\text{avgSim}_2 = \frac{\text{Sim}(m_3, m_5) + \text{Sim}(m_4, m_5)}{2} \tag{4}$$

$$\text{avgSim}_3 = \frac{\text{Sim}(m_6, m_5)}{1} \tag{5}$$

$$\text{avgSim}_4 = \frac{\text{Sim}(m_7, m_5)}{1} \tag{6}$$

These four average similarity indicate how the relationship among the class (m5) and other four classes. The higher value of the average similarity represents the stronger connection of the (m5) with that particular class. Let's assume that the value of avgSim3 is highest among all four and so, the class (m5) is merged with the class (m6) and final group of class has become *Group F: (m1,m2), (m3,m4), (m6, m5), (m7)* (Fig.13.).

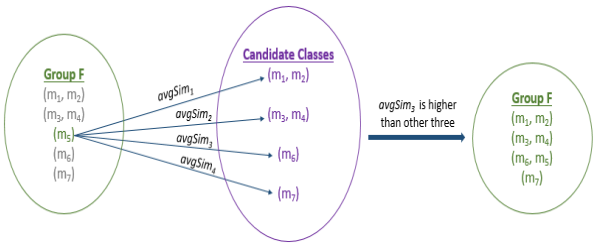


Fig.13. Merging (m5) with other classes

At this stage of merging procedure, since (m6) is no longer left with single method, it is not considered to be merged anymore. Thus, second iteration calculates the average similarly between (m7) and (m1,m2), (m3,m4), (m6, m5). Supposedly if the average similarly between (m7) and (m6, m5) is higher than the other two, these three methods are merged into a single class (m6, m5, m7). At this point, the final set of classes is *Group F: (m1,m2), (m3,m4), (m6, m5, m7)* (Fig.14.).

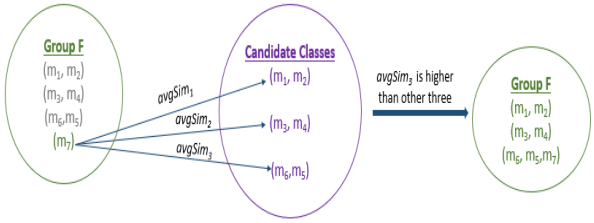


Fig.14. Merging (m7) with other classes

Initially, there were two sets of classes generated from the same methods *Group A: (m1,m2), (m3,m4,m5), (m6,m7)* and *Group B: (m1,m2), (m3,m4,m6), (m5,m7)*. Firstly, the classes of the both groups were compared thoroughly and resulted in a final cumulative set, *Group F: (m1,m2), (m3,m4), (m5), (m6), (m7)*. However, there were classes with single method in the final group and those were needed to be merged. These single method class are merged based on the similarity between the pairs of the methods. Therefore, the final result is *Group F: (m1,m2), (m3,m4), (m6, m5, m7).*

In this research, a God Class is first extracted into smaller classes using two different techniques. Firstly, the class is extracted based on the cohesion calculation and a set of clusters is produced as the candidate classes (section 3.1). Secondly, the same class is again extracted into a set of method clusters using HAC based on code documentation (section 3.2). From these two procedures, two sets of classes are produced having same methods. However, there is an issue of incorporating these two sets to produce a single set of classes which is more convenient. To address this issue, a cluster compositional algorithm is used [8] (section 3.3). This two phased approach first compares all the classes from both sets thoroughly and generates a cumulative set of classes. To avoid extracting classes with single method, in the second phase, classes with single method are merged with other classes from the set based on the similarity calculation between the methods. After merging all the single method classes, a final set of classes is generated which is more appropriate than the initial two sets. Each of the classes in the set represents a newly identified class which is needed to be extracted from the original God Class.

Table 4. C3 Value Comparison Shown on God Classes of *GanttProject*

Class	Pre-refactoring	Existing Approach		Proposed Approach	
		NOC	C3	NOC	C3
GanttOptions	0.18	3	0.27, 0.32, 0.36	2	0.82, 0.83
GanttProject	0.08	3	0.16, 0.36, 0.37	7	0.92, 0.99, 0.81, 0.96, 0.66, 0.94, 0.78
GanttGraphicArea	0.13	2	0.18, 0.15	3	0.62, 0.37, 0.44
GanttTree	0.14	2	0.22, 0.36	4	0.74, 0.62, 0.71, 0.97
GanttTaskPropertiesBean	0.13	2	0.18, 0.44	3	0.63, 0.46, 0.92
ResourceLoadGraphicArea	0.17	2	0.28, 0.35	2	0.63, 0.37
TaskImpl	0.27	3	0.31, 0.38, 0.41	2	0.93, 0.72

Table 5. Average C3 Value Comparison – Existing Approach vs. Proposed Approach

System	Pre-refactoring	Existing Approach	Proposed Approach
Xerces	0.11	0.23	0.70
Gantt	0.16	0.31	0.73
Average	**0.13**	**0.27**	**0.72**

3.4. Automatic Class Extraction

As mentioned earlier, following two distinct procedures (source code analysis and code documentation analysis), two sets of clusters are produced. Each of those clusters are supposed to be the new class and the elements of the cluster are supposed to be the methods of that class. After recommending the refactoring, the classes are also generated automatically in this study. The work flow of automatic class extraction is presented in Fig.16. The algorithm takes the source code of a God Class and the final set of method clusters as inputs. As the output of the procedure, new refactored classes are generated. The total number of clusters denotes how many classes are going to be constructed.

The extraction is performed through the *Algorithm* (Fig.15.). According to the algorithm, first a temporary list *variableList* is initialized with a null value (Line 1). All the clusters are transformed into new classes through an iterative process. At each iteration, all the class variables that are shared by the methods of a cluster (c) is found and stored in *variableList* (Line 3). Then, a new class is created for that cluster (Line 4). All the variables stored in *variableList* is declared at the very beginning of the new class (Line 5-7). Next, the constructor is generated for the class, and variables are added as its parameter (Line 8). The methods that are supposed to be in the new class are extracted from the source class and added in this class (Line 9-12). The procedure ends when all the classes are generated according to the newly identified clusters from the set.

Algorithm 1 Automatic Class Extraction

Input Source code of a God Class, C and a set of method clusters
Output New refactored classes

```
variableList ← ∅
for each method cluster, c ∈ C do
    variableList ← find and store all shared class variables by methods of c
    Create the new class
    for n ← 1 to variableList.length do
        Declare the variables from variableList
    end for
    Generate a constructor and add the variables as parameters
    for n ← 1 to c.length do
        Extract c[n] from C
        Add in the new class
    end for
end for
```

Fig.15. *Algorithm:* Automatic Class Extraction

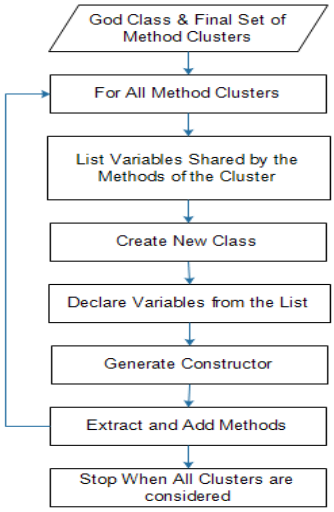


Fig.16. Work flow of Automatic Class Extraction

From section 3.3, a final set of clusters is formed. Each of the clusters denotes a new class and the elements of the cluster are the methods of that class. In this section, an algorithm is presented to extract the input God Class and generate the refactored classes. However, there is still scope of study how the other classes of the software project correlated with this God Class, need to be changed after the refactoring.

Table 6. Manual Inspection Based on Context - *AbstractDOMParser* and *DOMParserImpl*

Original Class	Refactored Classes		Notes
AbstractDOMParser	Class0	startGeneralEntity, endDTD, startDTD, endParameterEntity, endConditional, endAttlist, startExternalSubset, startParameterEntity, externalEntityDecl, internalEntityDecl, unparsedEntityDecl, getDocumentClassName, setDocumentClassName, startDocument, getDocument, reset, textDecl, processingInstruction,xmlDecl, doctypeDecl, characters,ignorableWhitespace, startCDATA, endDocument,endGeneralEntity, handleBaseURI, handleBaseURI, handleBaseURI, handleBaseURI, endExternalSubset,notationDecl, ignoredCharacters, elementDecl, startAttlist, abort, setLocale,	Total methods: 46 Misplaced groups: 3, methods: 3 (6.52%)
	Class1	startElement, createElementNode, createAttrNode, comment, emptyElement, endElement, endCDATA,startConditional, attributeDecl, setCharacterData,	
DOMParserImpl	Class0	parseURI, parse, abort, reset, getFilter, setFilter, setParameter, getParameter, getParameterNames, parseWithContext, dom2xmlInputSource	Total Methods: 15 Misplaced groups: 0, methods: 0 (0%)
	Class1	getAsync, getBusy	
	Class2	startElement, startElement	

Table 7. Manual Inspection Based on Context – *XincludeHandler*, *DurationImpl*, *AbstractSAXParser* and *BaseMarkupSerializer*

Original Class	Refactored Classes		Notes
XIncludeHandler	Class0	addUnparsedEntity, addNotation, checkAndSendNotation,checkNotation, checkAndSendUnparsedEntity,checkUnparsedEntity,modifyAugmentations, modifyAugmentations, modifyAugmentations, modifyAugmentations, getRecognizedFeatures,getRecognizedProperties, startDocument, isTopLevelIncludedItem, getResultDepth, setParent, isRootDocument, checkWhitespace,checkMultipleRootElements,setRootElementProcessed, getRootElementProcessed, saveBaseURI,restoreBaseURI, restoreLanguage, processXMLBaseAttributes, createInputSource	Total Methods: 84 Misplaced groups: 1, methods: 1 (1.19%)
	Class1	endAttlist,elementDecl,endConditional,endDTD, endExternalSubset, endParameterEntity,externalEntityDecl,notationDecl, startExternalSubset, startParameterEntity,unparsedEntityDecl,ignoredCharacters,internalEntityDecl, startAttlist, startConditional,startDTD, setDTDHandler, setDTDSource, attributeDecl, getDTDSource, getDTDHandler	
	Class2	hasXIncludeNamespace, isIncludeElement, isFallbackElement, processAttributes, saveLanguage,processXMLLangAttributes,	
	Class3	copyFeatures, copyFeatures,	
	Class4	getIncludeParentBaseURI, getIncludeParentLanguage, getSawFallback, getSawInclude, scopeOfBaseURI,getLanguage, getBaseURI, getPropertyDefault, sameBaseURIAsIncludeParent,escapeHref, getRelativeBaseURI, getIncludeParentDepth, setSawFallback, setSawInclude, getRelativeURI, isValidInHTTPHeader, sameLanguageAsIncludeParent	
	Class5	equals, equals, isDuplicate, isDuplicate, equals, equals, isDuplicate, isDuplicate, setProperty, searchForRecursiveIncludes	
	Class6	setFeature, getFeatureDefault	
DurationImpl	Class0	testNonNegative, testNonNegative, testNonNegative, testNonNegative, getFieldAsBigDecimal, sanitize, sanitize,sanitize, sanitize, getSign, calcSignum,wrap, isDigit,isDigitOrPeriod, parsePiece, organizeParts, parseBigInteger, parseBigDecimal, compare, compareDates, hashCode, toString, toString isSet, getField, getYears, getMonths, getDays, getHours, getMinutes, getSeconds, getInt, getTimeInMillis, getTimeInMillis, getCalendarTimeInMillis, getTimeInMillis, getTimeInMillis, addTo, addTo, normalizeWith, multiply, multiply, multiply, multiply, toBigInteger, add, subtract, negate, signum, addTo, addTo, writeReplace,	Total Methods: 54 Misplaced groups: 1, methods: 2 (3.70%)
	Class1	toString, toString	
AbstractSAXParser	Class0	startGeneralEntity, endGeneralEntity, startCDATA,endCDATA,endDocument, endParameterEntity,endExternalSubset,startParameterEntity,startExternalSubset, startDocument, xmlDecl,doctypeDecl, startElement,endElement, elementDecl, externalEntityDecl, notationDecl, unparsedEntityDecl,endDTD, parse, parse, parse, parse, getXMLVersion, getEncoding, getEncoding, getXMLVersion, getEncoding, getEncoding,setEntityResolver, getEntityResolver, setErrorHandler,getErrorHandler,setLocale,getFeature, setDTDHandler, setDocumentHandler,setContentHandler,getContentHandler, getDTDHandler, setFeature, setProperty,getProperty,setDeclHandler,getDeclHandler, setLexicalHandler, getLexicalHandler, reset, getPublicId, getSystemId, getLineNumber,getColumnNumber, getXMLVersion, getEncoding, getEncoding, setAttributes, endNamespaceMapping, startNamespaceMapping,	Total methods: 67 Misplaced groups: 1, methods: 2 (2.99%)
	Class1	characters, ignorableWhitespace, comment, processingInstruction, attributeDecl, internalEntityDecl	
	Class2	isDeclared, isDeclared, isDeclared,	
BaseMarkupSerializer	Class0	serialize, serialize, serialize, serializeElement,serialize,serialize,serialize,serialize, serialize,serialize,serializeNode,serializePreRoot,isDocumentState,getElementState, modifyDOMError,endDocument,checkUnboundNamespacePre_xedNode,content	Total methods: 30 Misplaced groups: 1, methods: 1 (3.33%)
	Class1	characters, characters	
	Class2	leaveElementState, enterElementState, getPrefix	
	Class3	getEntityRef,printText,printText,printHex, printEscaped, printEscaped	

Identifying meaningful and appropriate name for the newly extracted classes is another matter of concern. To do this, searching most frequent word in the class and name the class by that word was initially considered in this research. However, this idea does not lead to a relevant naming of class. The reason behind the failure of the idea is that the class names are not supposed to be contained in the class itself. Most of the cases, a class name is selected by the developers based on their own intuition and impression about the class. For Example, let's consider a class with the properties - head, hand, leg, eye etc. and methods - eat, sleep, work etc. It is highly expected that the name of the class should be Human or Man and there is hardly any possibility to find these words in the class. On the other hand, an extracted class after refactoring a god class (Table 6., row 2, Class0) is observed to detect most frequent word. Object (92 times), else (50 times), private (45 times) are at the top of the list of most used words. However, none of those is appropriate enough

to be considered as the name of a class. Therefore, naming the extracted classes is another field of research and hence kept out of the scope of this automated refactoring procedure.

The aim of this study is to emphasize on an aspect which can contribute in making improvements to refactoring procedure. To attain success in this purpose, contextual analysis on a God Class is proposed. In this approach, first an existing solution of cohesion based refactoring is implemented to refactor a God Class. Secondly, the same God Class is refactored based on contextual analysis. After that through a compositional algorithm, both the outcomes are integrated to produce a cumulative result. Thus, a final set of refactored classes is identified. Finally using an automatic class extraction method, those identified classes are extracted from the original God Class. The contribution of integrating context with cohesion is evaluated in the result analysis section to ensure that the performance of the refactoring technique is improved (Section 4).

4. Result Analysis

The aim of this section is to experimentally evaluate the performance of the proposed recommendation approach of God Class refactoring. For assessing the performance of this approach, it is applied on two well-known open source projects and the result is compared with the output of an existing system [3]. The proposed approach implemented this existing approach (Source Code Analysis) and integrated documented analysis with that. Therefore, the result of the existing technique is considered as the baseline for the comparative result analysis to show whether the result of the proposed approach is improved.

Table 8. Manual Inspection Based on Context – *DOMNormalizer* and *CoreDocumentImpl*

Original Class	Refactored Classes		Notes
DOMNormalizer	Class0	normalizeNode,normalizeDocument,setDocumentSource,getDocumentSource, addAttribute,normalizeAttributeValue	Total Methods: 28 Misplaced groups: 0, methods: 0 (0%)
	Class1	isCDataWF, isXMLCharWF, isCommentWF,isAttrValueWF,reportDOMError,	
	Class2	setAugmentations,endGeneralEntity,comment,characters,ignorableWhitespace, startCDATA,endCDATA,startDocument,xmlDecl,textDecl,doctypeDecl, processingInstruction,startGeneralEntity, endDocument	
	Class3	startElement, emptyElement, endElement,	
CoreDocumentImpl	Class0	cloneNode,cloneNode,cloneNode,cloneNode, clone,getTextContent, setTextContent, getDoctype, changes, getDocumentElement, getElementsByTagName, setStrictErrorChecking,getErrorChecking,changed,getStrictErrorChecking, getInputEncoding,getBaseURI,setInputEncoding, setXmlEncoding,setEncoding,load, getXmlEncoding,getEncoding,setVersion,getVersion,setXmlStandalone, setStandalone,getXmlStandalone,getStandalone,getDocumentURI, abort, normalizeDocument,getDomCon_g,setDocumentURI,getAsync,getMutationEvents, getNodeNumber,getNodeNumber,getElementById, clearIdentifiers,getNodeListCache, freeNodeListCache,isXML11Version,isXMLVersionChanged,setMutationEvents,	Total Methods: 127 Misplaced groups: 5 methods: 7 (5.51%)
	Class1	insertBefore, replaceChild, removeChild, getOwnerDocument,getNodeType,	
	Class2	createAttribute, createElement, createEntityReference, createProcessingInstruction, createEntity,createElementDe_nition, setXmlVersion,renameNode,setAsync, saveXML,isValidQName,createNotation,checkQName,getXmlVersion	
	Class3	createCDATASection,createComment, createTextNode,setUserData, setUserData, setUserData, setUserData,getUserData, getUserData,setUserDataTable, callUserDataHandlers, callUserDataHandlers,getFeature,getNodeNumber, getNodeNumber,	
	Class4	loadXML,importNode,importNode,importNode, importNode, getNodeName, createDocumentFragment, getImplementation,	
	Class5	adoptNode, isKidOK,removeUserDataTable,replacedText, getUserData, getUserData, modifyingCharacterData,modi_edCharacterData,insertingNode, insertedNode, removingNode,removedNode,replacingNode,replacedNode, replacingData, replacedCharacterData,setAttrNode, removedAttrNode,undeferChildren, callUserDataHandlers, callUserDataHandlers,deletedText,modifiedAttrValue, insertedText, renamedAttrNode,	
	Class6	putIdenti_er, removeIdentifier, getIdentifiers,renamedElement, getIdentifier,	
	Class7	createElementNS, createElementNS, createElementNS,createElementNS, createAttributeNS, createAttributeNS, createAttributeNS,createAttributeNS, createDocumentType, getElementsByTagNameNS,	

First of all, an evaluation is made after running it on the sample projects, and the accuracy of the approach is calculated using Conceptual Cohesion of Classes (C3) [9] and Lack of Cohesion of Methods (LCOM). These two metrics assess the cohesiveness among the classes and thus, it is ensured whether the newly identified classes are more cohesive than pre-refactored and refactored classes using existing technique [3]. Moreover, since the existing approach [3] used C3 and LCOM to calculate the cohesion, these are used to compare the performance of this proposed procedure. A comparative representation is shown using these three types of values (Section 4.1, 4.2).

The contextual similarities among the methods of the new refactored classes are inspected manually to ensure that the classes are contextually balanced. These results indicate that the proposed approach outperforms the existing one.

4.1. Comparative Result Analysis Using C3

As mentioned in the methodology section, an existing approach is implemented which refactors God Class using cohesion based grouping. This research intents to prove that integrating context based clustering along with the cohesion based clustering can improve the refactoring. To present the comparison, Conceptual Cohesion of Classes (C3) [9] is used. C3 is a conceptual cohesion metric which is used to compute the overlap of semantic information in a class in terms of textual similarity among methods [3]. The increased values of C3 proves that the cohesion of the class is also increased, and ensuring higher class cohesion is the motivation of the research. The C3 values of the classes before refactoring and after refactoring using both existing and proposed approach are calculated and compared in this section.

To calculate the Conceptual Cohesion of Classes (C3), ten classes from *Xerces (2.7.0)* [32] and seven from *GanttProject (1.10.2)* [36] are chosen. These classes are identified as God Class in the existing paper [3]. Based on C3, comparative analysis is performed on these classes.

In Table 1. and Table 3., the Class Name and Lines of Code (LOC) of the chosen ten God Classes of *Xerces (2.7.0)* and seven God Classes of *GanttProject (1.10.2)* are reported [3]. The LOC of those includes both source code and comments. The process of comment sections identification, selection and parsing is described briefly in section 3.2.1. The results achieved in this study is presented in terms of cohesion in Table 2. From the values of this table, it is clearly proved that for almost all classes, the cohesion is sensibly improved. In Table 2., the name of classes that are inspected is added in the first column (Class). Next, the C3 values of those classes before refactoring is stored (column Pre-refactoring) [3]. In the last two columns, the number of new classes (NOC) and their C3 values after refactoring is reported. First one adds the results using the existing approach (column Existing Approach) [3] and second one using the proposed approach (column Proposed Approach).

Table 9. Manual Inspection Based on Context – *DeferredDocumentImpl*

Original Class	Refactored Classes		Notes
DeferredDocumentImpl	Class0	createDeferredElement,createDeferredElement,createDeferredElement, createDeferredElement,createDeferredElement,createDeferredElement, createDeferredElement,createDeferredElement,createDeferredElement, setDeferredAttribute,setDeferredAttribute,setDeferredAttribute, setDeferredAttribute, setIdAttributeNode, setIdAttribute, getAttribute,setAttributeNode,createDeferredElementDenition	Total Methods: 116 Misplaced groups: 1, methods: 2 (1.72%)
	Class1	synchronizeData, putIdentier0, addElement,appendChild, setAsLastChild,putIdentier,getNodeIndex, removeAllElements,	
	Class2	createDeferredNotation,createDeferredEntity,setInputEncoding,size, getImplementation,getNamespacesEnabled,setEntityInfo, createNode,	
	Class3	getParentNode,getParentNode,getParentNode,getParentNode,getLastChild, getLastChild,getLastChild,getLastChild,getPrevSibling, getPrevSibling, getPrevSibling,getPrevSibling, getRealPrevSibling,getRealPrevSibling, getRealPrevSibling, getRealPrevSibling, getNodeName,getNodeName, getNodeName, getNodeName,getNodeValueString,getNodeValueString, getNodeValue,getNodeValue, getNodeValue, getNodeValue,getNodeExtra, getNodeExtra, getNodeExtra, getNodeExtra,getNodeType, getNodeType,getNodeType,getNodeType,getNodeURI,getNodeURI, getNodeURI, getNodeURI,elementAt,createDeferredDocument, createDeferredDocumentType,createDeferredEntityReference, createDeferredAttribute,createDeferredAttribute,createDeferredAttribute, createDeferredAttribute,createDeferredTextNode,createDeferredCDATASection, createDeferredProcessingInstruction,createDeferredComment, cloneNode, insertBefore,lookupElementDe_nition, getNodeValueString, getNodeValueString,	
	Class4	getNodeObject, synchronizeChildren, synchronizeChildren,synchronizeChildren, synchronizeChildren,synchronizeChildren,synchronizeChildren, synchronizeChildren,synchronizeChildren,synchronizeChildren,binarySearch, print, print, print, print	
	Class5	getTypeInfo, getNodeValue, getNodeValue, getChunkIndex, createChunk, setChunkIndex,clearChunkIndex,createChunk,ensureCapacity, ensureCapacity, ensureCapacity, ensureCapacity	

Table 10. LCOM Value Comparison Shown on God Classes of *Xerces*

Class	Pre-refactoring	Existing Approach		Proposed Approach	
		NOC	C3	NOC	C3
AbstractDOMParser	83	2	0, 0	2	40, 0
AbstractSAXParser	1126	3	49, 0, 451	3	19, 0, 5
BaseMarkupSerializer	921	2	27, 358	4	1, 0, 18, 0
CoreDocumentImpl	6,825	3	143, 190, 3322	8	15, 0, 87, 0, 11, 91, 0, 0
DeferredDocumentImpl	0	2	0, 41	6	1, 4, 36, 0, 0, 0
DOMNormalizer	456	2	66, 150	4	12, 0, 6, 0
DOMParserImpl	132	2	15, 54	3	1, 5, 31
DurationImpl	701	2	211, 355	2	47, 55
NonValidatingConfiguration	147	2	4, 82	2	3, 0
XIncludeHandler	4652	4	30, 602, 75, 188	7	34, 0, 121, 47, 0, 65, 20

For Instance, the C3 value of the God Class *AbstractDOMParser* was 0.21 before refactoring. Following the existing approach, the class is refactored in two new classes and the C3 values of those classes are 0.25 and 0.23. The proposed approach also refactored the God Class into two classes. By this approach, the C3 values of the two new classes are 0.64 and 0.47 and these values are higher than the previous two (row 1). In the case of *BaseMarkupSerializer* class, the existing approach refactored the class into two classes where the proposed approach divided the class into four new classes. However, the C3 values of new classes by proposed approach (0.73, 0.78, 0.54, 0.42) are higher than the C3 values of both pre-refactored class (0.08) and refactored classes using existing technique (0.18, 0.15) (row 3). If the all the rows of the table is observed closely, the evidence of the improvement of the result by proposed approach is clearly visible.

Next, the comparison of C3 values is performed on the selected seven God Classes of the project *GanttProject (1.10.2)*. In Table 4., The C3 values of the classes before refactoring (column 2) and after refactoing using both existing and proposed approach (column 3-4). The table shows how the result is improved if refactoring is performed using this approach. For example, C3 value of the class *GanttOptions* was 0.08 before rafctoring. Using the existing approach the class is divided into three classes with the C3 values - 0.27, 0.32 and 0.36. Although the class is split into two classes using proposed approach, the C3 values of the new classes become higher (0.82, 0.83) (row 1). The result of the rest six classes is also observed to be improved (row 2-7).

Table 11. LCOM Value Comparison Shown on God Classes of *GanttProject*

Class	Pre-refactoring	Existing Approach		Proposed Approach	
		NOC	C3	NOC	C3
GanttOptions	2100	3	1117, 295, 0	2	93, 0
GanttProject	2318	3	1233, 0, 0	7	40, 0, 69, 0, 19, 116, 17
GanttGraphicArea	845	2	511, 4	3	12, 21, 9
GanttTree	649	2	493, 15	4	3, 35, 21, 0
GanttTaskPropertiesBean	183	2	52, 4	3	18, 22, 4
ResourceLoadGraphicArea	252	2	143, 63	2	6, 29
TaskImpl	884	3	119, 58, 3	2	55, 3

Table 12. Average LCOM Value Comparison – Existing Approach vs. Proposed Approach

System	Pre-refactoring	Existing Approach	Proposed Approach
Xerces	1504	256	26
Gantt	1033	242	20
Average	**1310**	**257**	**23**

Table 5. compares the results achieved by the proposed and existing approach in terms of cohesion. For this, average of the cohesion values for both systems are calculated and compared. If the average C3 values of the table is observed closely, it is proved that the existing approach also achieves a sensible improvement of cohesion. However, this improvement is lower when compared with the proposed approach. The average cohesion values of before and after refactoring by both techniques are 0.13, 0.27 and 0.72 respectively (Table 5., row 3). Average C3 using proposed approach is more than five times better than the pre-refactoring and more than two times than the existing approach.

4.2. Comparative Result Analysis Using LCOM

The principal purpose of this research is to extract a God Class into smaller classes with higher cohesion. To ensure the success of the refactoring procedure it is necessary to prove that the cohesion of the refactored classes are higher than the cohesion of original God Class. In section 4.1, In terms of C3, the cohesion of the refactored classes are estimated and compared with both pre-refactored classes and refactored classes using an existing technique. In this section, another cohesion metric, Lack of Cohesion of Methods (LCOM) [37], is used to assess the performance of the proposed approach. It measures the lack of cohesion of a class. It is an inverse metric which implies that the higher the value of LCOM, the lower the class cohesion. It counts the sets of the methods that are not related in terms of local instance variables in the class [37].

The LCOM values of the classes before refactoring and after refactoring using both existing and proposed approach compared in this section. For this comparative analysis, the same ten classes from *Xerces (2.7.0)* [32] and seven from *GanttProject (1.10.2)* [36] are used which were chosen in C3 comparison (Section 4.1). These classes are identified as God Class in the existing paper [3]. In Table 1. and Table 3., the Class Name and Lines of Code (LOC) of the chosen seventeen classes are reported [3].

The results achieved in this study is presented in terms of LCOM in Table 10. and Table 11. From the values of these tables, it is clearly proved that for almost all the classes, the LCOM is decreased which shows that cohesion is sensibly improved. In both of the tables, the name of classes that are inspected is added in the first column (Class). Next, the LCOM values of those classes before refactoring is stored (column Pre-refactoring) [3]. In the last two columns, the number of new classes (NOC) and their LCOM values after refactoring is reported. First one adds the results using the existing approach (column Existing Approach) [3] and second one using the proposed approach (column Proposed Approach).

For Instance, in Table 10., the LCOM value of the God Class *AbstractSAXParser* was 1126 before refactoring. Following the existing approach, the class is refactored in three new classes and the LCOM values of those classes are 49, 0 and 451. The proposed approach also refactored the God Class into two classes. By this approach, the LCOM values of the two new classes are 19, 0, 5 and these values are lower than the previous two (row 2). In Table 11., in the case of *GanttProject* class, the existing approach refactored the class into three classes where the proposed approach divided the class into seven new classes. However, the LCOM values of new classes by proposed approach (40, 0, 69, 0, 19, 116, 17) are lower than the C3 values of both pre-refactored class (2318) and refactored classes using existing technique (1233, 0, 0) (row 2). If the all the rows of the both tables are observed closely, the evidence of the improvement of the result by proposed approach is clearly visible.

Table 12. compares the results measured using both the proposed and existing approach in terms of lack of cohesion among the methods in a class. For this, average of the cohesion values for both systems are calculated and compared. If the average LCOM values of the table is inspected, it is proved that cohesion lacking is already decreased in refactored classes using the existing approach. However, this improvement is higher in the proposed approach compared with the existing one. The average LCOM values of before and after refactoring by both techniques are 1310, 257 and 23 respectively (Table 12., row 3). The average LCOM after refactoring by proposed technique is lower than the other two average values.

The proposed God Class refactoring procedure is compared with an existing technique [3] using two different metrics: Conceptual Cohesion of Classes (C3) and Lack of Cohesion of Methods (LCOM). This analysis indicates that the newly identified class classes are more cohesive than the original God Classes and the refactored classes using existing technique.

4.3. Contextual Balance Inspection

In section 4.1 and 4.2, the effectiveness of the proposed approach is compared with existing approach in terms of C3 and LCOM. However, since one of the major contributions of this research is considering contextual aspect in refactoring God Class, it is highly needed to ensure that the modules of the refactored classes are contextually balanced. Therefore, a manual inspection is run on the nine God Classes of the project - *Xerces*. In order to perform this investigation, the names and descriptions of the methods of refactored classes are observed to capture the context of the methods. For this, two conditions are considered –

- Different actions on the identical data should be in the same class. For example, there are three methods: startElement, emptyElement, endElement. Since these three actions are performed on the same data element, these methods should be in the same class.
- Methods for same actions on different data should be in the same class. For example, the methods printText, printText, printHex, printEscaped, printEscaped performs print action on different data. Therefore, these five methods should be together.

A. Description of the Inspection

The nine God Classes of *Xerces* are first split into smaller classes following the proposed refactoring approach. Then, all methods of each refactored class are investigated to find out how many methods are misplaced based on their context (from Table 6. to Table 9.). The discussions on the classes are listed below –

AbstractDOMParser There were 46 methods in the original class. After refactoring into two new classes, three groups of methods are found to be misplaced based on context. Among those groups, three methods are misplaced. This implies that almost 6.52% (3 out of 46 methods) methods are not in the right class (Table 6.). The reasons behind the misplacement are –

- The actions start and end of CDATA should be in the same class. Group 1: (startCDATA) and (endCDATA) in Class0 and Class1 respectively. Methods Misplaced : 1
- Attribute declaration should be with the seven other declaration actions. Group 2: (externalEntityDecl,internalEntityDecl,unparsedEntityDecl,xmlDecl,doctypeDecl,notationDecl,elementDecl) in Class0 and (attributeDecl) in Class1. Methods Misplaced : 1
- The actions start and end of conditional should be in the same class. Group 3: (startcondtional) and (endconditional) in Class0 and Class1 respectively. Methods Misplaced : 1

DOMParserImpl In this class, initially there were 15 methods. After reafctoring into three new classes, no method is found to be misplaced on the basis of contextual similarities (Table 6.).

XIncludeHandler 1.19% (1 out of 84) methods are not in the right place (Table 7.)

- The actions save, restore and get on BaseURI should be together. Group 1: (saveBaseURI, restoreBaseURI) in Class0 and (getBaseURI) in Class4. Methods Misplaced: 1

DurationImpl 2 out of 54 methods (3.70%) are not placed perfectly (Table 7.).

- Same method names with different signatures should be in the same class. Group 1: Two (toString) methods in Class0 and two (toString) methods in Class1. Methods Misplaced : 2

AbstractSAXParser Out of 67, 2 methods (2.99%) are found being misplaced after refactoring (Table 7.).

- All the declaration functions should be together. Group 1: (xmlDecl, doctypeDecl, elementDecl, externalEntityDecl,notationDecl, unparsed- EntityDecl) in Class0 and (attributeDecl, internal- EntityDecl) in Class1. Methods Misplaced : 2

BaseMarkupSerializer 3.33% (2 out of 30) methods are not in the right place on the basis of context (Table 7.).

- All actions on element state should be in the same class. Group 1: (getElementState) in Class0 and (leaveElementState, enterElementState) in Class2. Methods Misplaced : 1

DOMNormalizer No method is found misplaced on the basis of contextual similarities (Table 8.).

CoreDocumentImpl After refactoring five groups of methods and among those groups seven out of 127 methods (5.51%) are misplaced (Table 8.).

- The actions clear and get on identifiers should be together. Group 1: (clearIdentifiers) and (getIdentifiers) in Class0 and Class6 respectively. Methods Misplaced : 1
- Same methods with different signature should be together. Group 2: Two (getNodeNumber) methods in Class0 and two (getNodeNumber) methods in Class3. Methods Misplaced : 2
- Same methods with different signatures should be together. Group 3: Two (getUserData) methods in Class3 and two (getUserData) methods in Class5. Methods Misplaced : 2
- The actions load and save on XML should be together. Group 4: (saveXML) and (loadXML) in Class2 and Class4. Methods Misplaced : 1
- The actions set and remove on userDataTable should be together. Group 5: (setUserDataTable) and (removeUserDataTable) in Class3 and Class5. Methods Misplaced : 1

DeferredDocumentImpl 1.72% (2 out of 30) methods are misplaced on the basis of context (Table 9.).

- Same methods with different signatures should be together. Group 1: Four (getNodevalue) methods in Class0 and two (getNodeValue) methods in Class3. Methods Misplaced : 2

B. Discussion on Result

A thorough manual inspection is performed on the nine predefined God Classes of *Xerces(2.7.0)* [32]. The purpose of this inspection was to ensure that the methods are contextually balanced after refactored using the proposed approach. For this, the context of the methods of a particular refactored class are observed and compared with each other. Thus, the method sets are found which are misplaced according to their context. Based on the description of the last section (from Table 6. to Table 9.), the number of the misplaced methods is very low with the respect of total methods of a God Class. The range of this misplacement is from 0% to 6.52% which implies that on average almost 2.78% methods are not placed correctly. Although these methods are split in different classes after refactoring, these methods are actually supposed to be together. However, most of the methods (on average 97.23%) are placed correctly on basis of their context.

This section has described the results of the proposed approach of God Class refactoring. The approach has been applied on two projects to analyze the effectiveness of the approach. The analysis was demonstrated in two different phases. Firstly, a comparative analysis is performed on the projects in terms of C3 and LCOM to ensure that the cohesion of classes are increased after refactoring. The comparisons shows that if the God Classes are refactored following the proposed approach, the cohesion of the newly identified classes are sensibly higher than both before and after refactoring using an existing approach. In the next phase, the context of the methods of the new classes are investigated manually and reported that a very low number of methods (on average 2.78%) are misplaced in respect of their context.

5. Threats to Validity

The main threats that might influence the result of this study are discussed in this section. In this research, while proposing the refactoring approach only cohesion was considered. However, emphasizing on different aspect (e.g., coupling or complexity) in refactoring can represent different scenario.

Another issue in result analysis is that the comparative analysis is performed only on two open source system (*Xerces (2.7.0)* [32] and *GanttProject (1.10.2)* [36]) and thus, the result cannot be generalized. The issue was raised due to unavailability of projects for comparison in the existing approach [3]. However, the proposed approach can be applied on any God Class of any software system.

6. Conclusion

In this paper, a God Class refactoring recommendation and automatic extraction technique is proposed. This technique splits a God Class into smaller classes to improve the design quality of the system in terms of both cohesion and context. This technique can assist developers in making their maintenance activity easier. Although an affluent number of researches have been conducted in this arena, to the best of authors' knowledge, no existing approach gives emphasis on the contextual aspect in refactoring God Class. Therefore, considering context of a God Class in refactoring is proposed to add significant contribution so that the work can advance the field of God Class refactoring from the present state of knowledge. The purpose of this study is to figure out how the result is improved in comparison with existing works if another aspect, contextual analysis, is incorporated.

A refactoring approach is proposed which generates the clusters of the methods based on their code documentation. To incorporate both cohesion and contextual analysis, a cluster compositional algorithm is used which generates the final set of method groups. Each method group of this set is supposed to be a new class. Therefore, the newly identified classes are extracted automatically from the source class and constructed as individual classes. To provide a proper justification of this research, an extensive result analysis is designed with both comparative analysis and manual inspection. Using two cohesion metrics, the comparative result analysis is provided where it is shown that the result becomes better if the context is added in the refactoring process. The technique is run on two open source softwares and the result is compared with the output of an existing system. In terms of Conceptual Cohesion of Classes (C3), the average cohesion of the proposed technique is 0.72 which is more than five times higher than pre-refactoring (0.13) and more than two times higher than the existing system (0.27). In terms of Lack of Cohesion of Methods (LCOM), the average LCOM in the refactored classes using proposed approach is 23 which is lower than both the pre-refactored classes (1,310) and the refactored classes using existing system (257). Moreover, contextual similarity among the methods of the refactored classes is manually inspected. The percentage of method misplace is also very low (on average 2.78%).

In the proposed refactoring approach, both cohesion and contextual similarity based factors have the same priority in the similarity measurement process while composing the clusters. The future plan is to assign different weights on the two similarity factors in order to analyze which set of weights results better. This research only construct the body of refactored classes. However, there must be some other classes which use or access the properties and functionalities of the original class. When the class is divided into smaller classes, changes are supposed to be required in those corresponding classes. Therefore, adopting those required changes is another future work direction of this research.

After extraction, identifying meaningful and appropriate names for the newly extracted classes is also another matter concern. As discussed earlier, searching most frequent word in the class and name the class by that word was the initial approach attempted to address this issue in this research. However, this idea does not lead to a relevant naming of class. The reason behind the failure of the idea is that the class names are not supposed to be contained in the class itself. Most of the cases, a class name is selected by the developers based on their own intuition and impression about the class. Therefore, naming the extracted classes is kept out of the scope of the objective of this study and can be explored as a dimension of future research.

Acknowledgment

This research is supported by the fellowship from ICT Division, Ministry of Posts, Telecommunications and Information Technology, Bangladesh.

[56.00.0000.028.33.094.18-168 Dated 03.05.2018]

References

[1] Fowler, M. and Beck, K. (1999) *Refactoring: improving the design of existing code.* Addison-Wesley Professional, Boston, MA, USA.

[2] Bavota, G., De Lucia, A., Marcus, A., and Oliveto, R. (2010) A two-step technique for extract class refactoring. *Proceedings of the IEEE/ACM international conference on Automated software engineering*, Antwerp, Belgium, 20-24 September, pp. 151{154. ACM, New York, NY, USA.

[3] Bavota, G., De Lucia, A., Marcus, A., and Oliveto, R. (2014) Automating extract class refactoring: an improved method and its evaluation. *Empirical Software Engineering*, 19, 1617-1664.

[4] Gethers, M. and Poshyvanyk, D. (2010) Using relational topic models to capture coupling among classes in object-oriented software systems. *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, Timisoara, Romania, 12-18 September, pp. 1-10. IEEE Computer Society,Washington, DC, USA.

[5] Bavota, G., Oliveto, R., De Lucia, A., Antoniol, G., and Gueheneuc, Y.-G. (2010) Playing with refactoring: Identifying extract class opportunities through game theory. *Software Maintenance (ICSM), 2010 IEEE International Conference on*, Timisoara, Romania, 12-18 September, pp. 1-5. IEEE.

[6] Fokaefs, M., Tsantalis, N., Stroulia, E., and Chatzigeorgiou, A. (2011) Jdeodorant: identification and application of extract class refactorings. *Proceedings of the 33rd International Conference on Software Engineering*, Honolulu, HI, USA, 21-28 May, pp. 1037-1039. IEEE.

[7] Bavota, G., Gethers, M., Oliveto, R., Poshyvanyk, D., and Lucia, A. d. (2014) Improving software modularization via automated analysis of latent topics and dependencies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23, 4:1-4:33.

[8] Jeba, T., Uddin Mahmud, T. S., and Nahar, N. (2018) A cluster compositional algorithm for incorporation of multiple sets of clusters of identical data. *2018 Joint 7th International Conference on Informatics, Electronics Vision (ICIEV) and 2018 2nd International Conference on Imaging, Vision Pattern Recognition (icIVPR)*, Kitakyushu, Japan, 25-29 June, pp. 59-64. IEEE.

[9] Marcus, A., Poshyvanyk, D., and Ferenc, R. (2008) Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34, 287-300.

[10] Abilio, R., Padilha, J., Figueiredo, E., and Costa, H. (2015) Detecting code smells in software product lines – an exploratory study. *2015 12th International Conference on Information Technology-New Generations*, Las Vegas, NV, USA, 13-15 April, pp. 433{438. IEEE.

[11] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D., and De Lucia, A. (2014) Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, 41, 462-489.

[12] Mansoor, U., Kessentini, M., Maxim, B. R., and Deb, K. (2017) Multi-objective code-smells detection using good and bad design examples. *Software Quality Journal*, 25, 529-552.

[13] Kessentini, W., Kessentini, M., Sahraoui, H., Bechikh, S., and Ouni, A. (2014) A cooperative parallel search-based software engineering approach for code smells detection. *IEEE Transactions on Software Engineering*, 40, 841-861.

[14] Bavota, G., De Lucia, A., and Oliveto, R. (2011) Identifying extract class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software*, 84, 397-414.

[15] Akash, P., Sadiq, A., and Kabir, A. (2019) An approach of extracting god class exploiting both structural and semantic similarity. *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE*, Heraklion, Crete, Greece, 4-5 May, pp. 427-433. SciTePress.

[16] Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003) Latent dirichlet allocation. *Journal of Machine Learning research*, 3, 993-1022.

[17] Tan, P.-N. et al. (2006) *Introduction to data mining*. Pearson Education India, Boston, MA, USA.

[18] Jain, A. K. and Dubes, R. C. (1988) *Algorithms for Clustering Data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[19] Mazinanian, D., Tsantalis, N., Stein, R., and Valenta, Z. (2016) Jdeodorant: clone refactoring. *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on*, pp. 613-616. IEEE.

[20] Tsantalis, N., Chaikalis, T., and Chatzigeorgiou, A. (2008) Jdeodorant: Identification and removal of type-checking bad smells. *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, Athens, Greece, 1-4 April, pp. 329-331. IEEE.

[21] Fokaefs, M., Tsantalis, N., and Chatzigeorgiou, A. (2007) Jdeodorant: Identification and removal of feature envy bad smells. *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, Paris, France, 2-5 October, pp. 519-520. IEEE.

[22] Larson, R. R. (2010) Introduction to information retrieval. *Journal of the American Society for Information Science and*

Technology, 61, 852-853.
[23] Fokaefs, M., Tsantalis, N., Chatzigeorgiou, A., and Sander, J. (2009) Decomposing object-oriented class modules using an agglomerative clustering technique. *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, Edmonton, AB, Canada, 20-26 September, pp. 93-101. IEEE.
[24] Dexun, J., Peijun, M., Xiaohong, S., and Tiantian, W. (2013) Detection and refactoring of bad smell caused by large scale. *International Journal of Software Engineering & Applications*, 4, 1.
[25] Chang, J., Blei, D. M., et al. (2010) Hierarchical relational models for document networks. *The Annals of Applied Statistics*, 4, 124-150.
[26] Kanungo, T., Mount, D. M., Netanyahu, N. S., Piatko, C. D., Silverman, R., and Wu, A. Y. (2002) An efficient k-means clustering algorithm: Analysis and implementation. *IEEE transactions on pattern analysis and machine intelligence*, 24, 881-892.
[27] Bentley, J. L. (1975) Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18, 509-517.
[28] Parsian, M. (2015) *Data algorithms: recipes for scaling up with Hadoop and Spark*. O'Reilly Media, Inc., Sebastopol, California.
[29] Fokaefs, M., Tsantalis, N., Stroulia, E., and Chatzigeorgiou, A. (2012) Identification and application of extract class refactorings in object-oriented systems. *Journal of Systems and Software*, 85, 22412260.
[30] Anquetil, N. and Lethbridge, T. C. (1999) Experiments with clustering as a software remodularization method. *Sixth Working Conference on Reverse Engineering (Cat. No. PR00303)*, Atlanta, GA, USA, USA, 8-8 October, pp. 235-255. IEEE.
[31] AbdAllah, L. and Shimshoni, I. (2014) Mean shift clustering algorithm for data with missing values. *International Conference on Data Warehousing and Knowledge Discovery*, Munich, Germany, 2-4 September, pp. 426-438. Springer.
[32] Xerces-J 2 7 0. https://github.com/apache/xerces2-j/releases/tag/Xerces-J_2_7_0. Online; accessed 10 February, 2018.
[33] Baeza-Yates, R., Ribeiro-Neto, B., et al. (1999) *Modern information retrieval*. ACM press, New York.
[34] Palomba, F., Panichella, A., De Lucia, A., Oliveto, R., and Zaidman, A. (2016) A textual-based technique for smell detection. *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pp. 1-10. IEEE.
[35] Porter, M. F. (1980) An algorithm for suffix stripping. *Program*, 14, 130-137.
[36] GanttProject. https://sourceforge.net/projects/ganttproject/ files\%2FOldFiles/. Online; accessed 13 February, 2018.
[37] Li, W. and Henry, S. (1993) Maintenance metrics for the object oriented paradigm. *[1993] Proceedings First International Software Metrics Symposium*, Baltimore, MD, USA, USA, 21-22 May, pp. 52-60. IEEE.

Authors' Profiles

Tahmim Jeba has completed Master of Science in Software Engineering (MSSE) and Bachelor of Science in Software Engineering (BSSE) degree from the Institute of Information Technology, University of Dhaka. Her research interests include Software Engineering, Requirement Engineering, Software Design and Quality.

Tarek Mahmud is a PhD student in the Department of Computer Science at Texas State University. He is working in the Software Engineering lab there. He completed his Bachelor of Science in Software Engineering (BSSE) from Institute of Information Technology, University of Dhaka. His area of interest is Software Engineering especially in program analysis and software validation, verification and testing.

Pritom S. Akash is a graduate student in Institute of Information Technology, University of Dhaka, where he is pursuing a certificate in Master of Science in Software Engineering (MSSE). He also completed Bachelor of Science in Software Engineering (BSSE) from the same institute. His research interests include Machine Learning, Data Mining and Software Engineering.

Nadia Nahar is Lecturer at the Institute of Information Technology (IIT), University of Dhaka, Bangladesh. She pursued her Master of Science in Software Engineering (MSSE) and Bachelor of Science in Software Engineering (BSSE) from the same institution. She was the gold medalist for attaining top score in her class. As a student, her efforts have earned awards from different national and international software and programming competitions, project showcasings as well as publications in various international conferences. She has the experiences of working both in industry and academia. Her core areas of interest are software engineer ing, web technologies, systems and security.

How to cite this paper: Tahmim Jeba, Tarek Mahmud, Pritom S. Akash, Nadia Nahar, "God Class Refactoring Recommendation and Extraction Using Context based Grouping", International Journal of Information Technology and Computer Science(IJITCS), Vol.12, No.5, pp.14-37, 2020. DOI: 10.5815/ijitcs.2020.05.02

